VectraOps

Knowledge Library

CHECKLIST

Node.js Production Readiness Checklist

A practical Node.js production readiness checklist for verifying security, reliability, observability, and release controls before you ship to production.

Programming - July 29, 2026

Purpose

A Node.js app can work in development and still fail in production because of missing dependency controls, weak error handling, incomplete observability, or unsafe deployment defaults. This checklist helps you verify the operational basics before release so you can catch avoidable failures, reduce security exposure, and confirm that the service can be supported after go-live.

Use this checklist to decide whether the current build is ready for production, what evidence to collect, and which gaps still need to be closed.

How to use this checklist

Work through the phases in order. For each item, confirm the control, capture evidence, and record the owner who can fix or approve it. Treat any failed item as a release blocker unless you have a documented exception with a time-bound remediation plan.

A good review is not just a yes/no exercise. It should show what was checked, how it was verified, and what changed as a result.

Phase 1: Application baseline and runtime assumptions

This phase confirms that the service starts consistently, the runtime requirements are explicit, and the app behaves predictably under the Node.js version you intend to run.

Checklist items

- ? Confirm the supported Node.js version range is documented and matches the target runtime in development, CI, staging, and production.
- ? Confirm package.json declares the correct engines field, if your deployment process relies on it.
- ? Review startup logs to verify the service binds to the expected host and port without manual intervention.
- ? Validate that required environment variables are documented and that missing values fail fast with clear errors.
- ? Test that configuration is loaded from the intended source and that defaults do not silently override production settings.
- ? Confirm the app exits non-zero when a critical dependency, secret, or config value is missing.
- ? Document the minimum file system, network, and process permissions required by the service.
- ? Assign ownership for runtime compatibility checks when upgrading Node.js or changing deployment images.

Evidence to capture

Record the Node.js version used in each environment, the startup command, the resolved config values, and a log excerpt from a clean boot. Capture a copy of any runtime requirements doc or deployment note that the operator can use during release.

Acceptance criteria

The service starts without manual fixes, required settings are explicit, and the runtime version is known to be supported. Any hidden dependency on local files, developer shell variables, or ad hoc startup steps should fail this phase.

Owner and review cadence

Primary owner: application engineer. Review cadence: every release and after every Node.js version change.

Common mistakes

A common failure is assuming local .env behavior matches production. Another is allowing the app to start with placeholder values that later cause partial outages or accidental connections to the wrong backend.

Phase 2: Dependency and supply chain review

This phase checks whether the dependency tree is trustworthy, reproducible, and monitored for risk. If your application includes express-based APIs, this is also the right time to verify that your route layer and auth middleware are not introducing unnecessary package sprawl; if you are building that layer from scratch, this REST API implementation pattern can help you keep the surface area deliberate.

Checklist items

- ? Confirm package-lock.json, npm-shrinkwrap.json, or the approved lockfile format is committed and used in CI.
- ? Review dependency sources to verify packages come from the approved registry or mirror.
- ? Validate that direct dependencies are limited to what the service actually uses.
- ? Test the install process from a clean workspace to confirm reproducible dependency resolution.
- ? Confirm dependency scanning runs in CI and produces an auditable result.
- ? Review alerts or findings for high-risk packages, transitive dependency changes, and abandoned modules.
- ? Document the approval rule for introducing new dependencies, including security review and maintainership checks.
- ? Assign a clear owner for dependency remediation and lockfile updates.

Evidence to capture

Capture the lockfile, the CI install log, the dependency scan report, and the package review notes for any newly introduced module. If a package was replaced or removed, keep the rationale and change record together.

Acceptance criteria

Dependency installs are reproducible, security review is active, and no unapproved package is required for runtime. Any finding that affects trusted code execution, authentication, or request handling should block release until resolved or formally accepted.

Owner and review cadence

Primary owner: application engineer with security review input. Review cadence: every pull request that changes dependencies and every release candidate.

Common mistakes

Teams often accept transitive dependencies without checking whether the package is still maintained. Another frequent issue is updating packages only in one environment and then discovering lockfile drift during deployment.

Phase 3: Security controls and secret handling

This phase verifies that the app protects secrets, validates input, and applies security controls consistently in production. If your service exposes HTTP endpoints, rate control is one of the practical controls to verify alongside auth and logging; for an implementation reference, see Node.js API Rate Limiting with Redis and Express Middleware.

Checklist items

- ? Confirm secrets are stored outside source control and injected through an approved secret management mechanism.
- ? Review the codebase to verify no credentials, API keys, tokens, or private certificates are committed.
- ? Validate that input validation rejects malformed payloads before business logic executes.
- ? Test that authentication and authorization checks apply to every protected route, job, or administrative action.
- ? Confirm cookie, token, and session settings match the deployment model and security policy.
- ? Review HTTP security headers, TLS termination assumptions, and any proxy trust configuration that affects request security.
- ? Validate that error messages do not expose secrets, stack traces, or internal system details to callers.

- ? Document the incident response path for leaked secrets, including revocation and rotation steps.

Evidence to capture

Capture the secret source-of-truth reference, the validation results for protected routes, the security review notes, and sample sanitized error output. Keep proof that the repository scan completed with the current commit.

Acceptance criteria

Secrets are not embedded in code, access controls are enforced on the intended paths, and external error responses do not leak sensitive data. If any secret is discovered in source history or logs, treat the build as not ready until rotation and cleanup are complete.

Owner and review cadence

Primary owner: security engineer or application owner. Review cadence: every release and immediately after any secret rotation or auth change.

Common mistakes

A typical mistake is validating only the happy path and missing an unprotected admin endpoint. Another is relying on front-end checks while leaving backend routes accessible without server-side authorization.

Phase 4: Reliability, error handling, and failure behavior

This phase checks whether the service fails safely, reports errors clearly, and avoids turning local faults into cascading outages.

Checklist items

- ? Test startup failure behavior to verify the process exits cleanly when a required service is unavailable.
- ? Validate that uncaught exceptions and unhandled promise rejections are captured and logged.
- ? Review retry logic to confirm it is bounded and does not create infinite retry loops.
- ? Confirm timeouts exist for outbound HTTP, database, and queue calls.
- ? Validate graceful shutdown handling for SIGTERM or the equivalent container termination signal.
- ? Test that in-flight requests are drained or rejected consistently during shutdown.
- ? Review circuit-breaking or backoff behavior if the app depends on fragile downstream services.
- ? Document manual recovery steps for known failure modes.

Evidence to capture

Capture logs from a controlled downstream failure, the timeout settings, and the shutdown behavior during a deployment or container stop. Record whether the process exits, retries, or degrades as expected.

Acceptance criteria

The app does not hang indefinitely, does not retry without bounds, and does not corrupt state during shutdown. Known downstream failures should produce predictable logs and a recoverable service state.

Owner and review cadence

Primary owner: application engineer. Review cadence: each release candidate and after changes to data access, queues, or outbound integrations.

Common mistakes

Many teams only test normal traffic and never verify how the service behaves when the database is slow or unavailable. Another common error is using a graceful shutdown handler that logs correctly but does not actually stop new work from entering the process.

Phase 5: Observability and operational evidence

This phase confirms that operators can see what the service is doing, diagnose failures, and prove whether the release is healthy.

Checklist items

- ? Confirm logs contain a consistent request identifier or trace context where practical.
- ? Review log fields to verify they are structured enough for search and alerting.
- ? Validate that success, warning, and error paths produce the expected log level and message.
- ? Test that key service metrics are emitted, including request volume, latency, and error rate where applicable.
- ? Confirm health checks distinguish between liveness and readiness, if your platform supports both.
- ? Review alert thresholds to ensure they align with operational impact rather than raw noise.

- ? Document where logs, metrics, and traces are stored and who can access them.
- ? Assign an on-call owner who can interpret the telemetry during an incident.

Evidence to capture

Keep sample logs from a request, a failure, and a startup event. Capture dashboard names, metric names, or alert rules used to judge health. Include the health check endpoints or probe configuration if they exist.

Acceptance criteria

A responder can identify what failed, when it failed, and whether the failure is isolated or systemic. If the service cannot be diagnosed from telemetry within a reasonable time, observability is not production ready.

Owner and review cadence

Primary owner: operations or platform engineer. Review cadence: every release and after any change to logging, metrics, tracing, or alert rules.

Common mistakes

A frequent problem is logging too little during errors and too much during normal traffic. Another is treating a green health check as proof that the app is actually serving requests successfully.

Phase 6: Deployment, rollback, and release controls

This phase verifies that the deployment path is repeatable and that you can recover quickly if the release causes a regression.

Checklist items

- ? Confirm the deployment method is documented and repeatable by someone other than the original author.
- ? Validate that configuration differences between environments are explicit and version-controlled where appropriate.
- ? Test that a rollback restores the previous known-good version without manual rework.
- ? Review deployment health checks to confirm they detect failed start, crash loops, and bad readiness states.
- ? Confirm database migrations, if used, have a rollback or compatibility strategy.
- ? Document the release approval gate and the conditions that block promotion to production.
- ? Validate that artifact versions, build hashes, or image tags can be traced back to source control.
- ? Assign responsibility for production deployment approval and rollback execution.

Evidence to capture

Capture the deployment runbook, the rollback procedure, the artifact version identifier, and evidence from a staging deployment or dry run. Keep notes on any manual steps that are still required.

Acceptance criteria

The release can be deployed and rolled back predictably, and the team knows what happens if the build fails after promotion. Any undocumented manual step in the critical path is a sign that the process still needs hardening.

Owner and review cadence

Primary owner: release owner or DevOps engineer. Review cadence: every release and after any change to build or deployment automation.

Common mistakes

Teams often validate only forward deployment and skip rollback testing. Another error is assuming migration scripts are safe because they ran successfully once in staging.

Phase 7: Capacity, performance, and environment fit

This phase checks whether the service can handle the expected workload and whether the target environment matches the app?s resource profile.

Checklist items

- ? Confirm the service has explicit CPU, memory, and file descriptor expectations documented.
- ? Test baseline performance under a representative load profile before approving production use.
- ? Validate that memory usage stays within the expected range during steady traffic and burst traffic.
- ? Review whether the event loop, worker model, or queue handling introduces known bottlenecks.
- ? Confirm concurrent request limits or backpressure controls are in place where needed.
- ? Validate that container, VM, or host resource limits match the app?s actual needs.
- ? Document how to recognize resource exhaustion before it becomes a full outage.

- ? Assign ownership for performance regression review after significant code or dependency changes.

Evidence to capture

Record the test profile, resource settings, and a comparison between expected and observed resource usage. Include any tuning changes made during validation.

Acceptance criteria

The service performs within the approved operating envelope, and resource limits are known and intentional. If performance depends on optimistic assumptions about traffic shape or hardware size, the deployment is not yet safe.

Owner and review cadence

Primary owner: application engineer or performance owner. Review cadence: before production launch, after major traffic changes, and after significant dependency or runtime upgrades.

Common mistakes

A common mistake is testing with synthetic traffic that is too small or too uniform to expose bottlenecks. Another is setting platform limits without confirming the app can recover gracefully when those limits are reached.

Readiness scoring

Use the following simple scoring method after completing the checklist:

- 2 points: confirmed and evidenced

- 1 point: partially confirmed or evidence is incomplete
- 0 points: not confirmed

Score each phase, then total the result.

- 12 to 14 points: ready for production with normal monitoring
- 8 to 11 points: conditionally ready after documented fixes
- 0 to 7 points: not ready for production

If a failed item affects security, identity, data integrity, or rollback capability, treat it as a blocker even if the total score looks acceptable.

Pass/fail decision rules

Use these rules to make the release decision consistent:

- Pass only when every critical control is verified and the evidence is available.
- Fail when any required secret, auth check, or rollback step is missing or untested.
- Fail when the app starts only through manual intervention or undocumented operator knowledge.
- Conditionally pass only when a non-critical gap has a dated remediation plan and an accountable owner.

Final review record

Document the result in a short release note that includes the date, reviewer, overall score, remaining gaps, and any exception granted. If the service passed, keep the evidence set with the release artifact so future changes can be compared against the same standard. If it failed, preserve the checklist output and close the gaps before resubmitting for production approval.

Use this guidance together with ASP.NET programming and Kafka Streams security to connect the workflow with related operational context already available on the site.